NPS-CS-95-004

NAVAL POSTGRADUATE SCHOOL
Monterey, California

UNIFORM REPRESENTATION OF DATA TYPES IN POLYMORPHIC C

by

Carl M. Pederson, Jr., CDR, USN

October 1995

Approved for public release; distribution is unlimited

Prepared for: Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral M. J. Evans
Superintendent

Richard Elster
Provost

This report was prepared for a directed study course (CS 4800) titled Advanced Topics in Compilation.

Carl M. Pederson Jr.
CDR Carl M. Pederson, Jr., USN

Reviewed by:

D. Volpano
DENNIS VOLPANO
Assistant Professor
of Computer Science

Released by:

P. J. Marto
PAUL J. MARTO
Dean of Research

T. D. Lewis
TED LEWIS
Chairman
Department of Computer Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPSCS-95 -004	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Uniform Representation of Data Types in Polymorphic C.

12. PERSONAL AUTHOR(S): CDR Carl M. Pederson Jr., USN

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 7/95 TO 10/95	October 1995	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	polymorphism, C programming language

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A polymorphic dialect of C, called Polymorphic C, has been proposed. The dialect retains the flexibility of C while incorporating ML-style polymorphism and rigorous type reconstruction. Supporting polymorphism in a programming language often requires sacrificing either speed, space, or both in the executable code. The preferred implementation of Polymorphic C would preserve the speed and space efficiency of C. This paper demonstrates an approach for generating efficient executable code for Polymorphic C based on a variation of uniform representation and using byte-wise manipulation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: CDR Carl M. Pederson Jr., USN	22b. TELEPHONE (Include Area Code): (408) 375 - 5628; 22c. OFFICE SYMBOL: CS

Uniform Representation of Data Types in Polymorphic C

Abstract

A polymorphic dialect of C, called *Polymorphic C*, has been proposed. The dialect retains the flexibility of C while incorporating ML-style polymorphism and rigorous type reconstruction. Supporting polymorphism in a programming language often requires sacrificing either speed, space, or both in the executable code. The preferred implementation of Polymorphic C would preserve the speed and space efficiency of C. This paper demonstrates an approach for generating efficient executable code for Polymorphic C based on a variation of *uniform representation* and using byte-wise manipulation.

1. Introduction

Providing the widely used imperative language C with polymorphism could dramatically increase the reuse potential of C programs. A polymorphic dialect of C, called *Polymorphic C* (abbreviated Poly C in this paper) has been proposed by Smith and Volpano [SmVo95]. The dialect retains the flexibility of C while incorporating ML-style polymorphsim and rigorous type reconstruction. The acceptance of the dialect requires an efficient implementation. The compiled code must come close to the speed and space efficiency of C.

Incorporating polymorphism in a programming language is generally incompatible with fast execution speed and low space overhead. Additionally polymorphism introduces various problems into the process of generating intermediate code during compilation. Many implementation techniques addressing these problems have been proposed for functional languages [MDCB91], [Ler92], [ShAp95], [Thi95]. These approaches could be implemented in a compiler for Poly C, but not without adversely affecting executable code. This is because function calls are used to perform explicit coercion between monomorphic types and a uniform representation used by polymorphic functions (wrap

and unwrap). This paper describes an implementation approach for Poly C that preserves the speed efficiency with only a modest storage penalty.

The proposed implementation employs C code as the intermediate language. A monomorphic program is translated directly to C code. For polymorphic functions, a uniform and strict calling convention is used to allow different data types to be passed and returned from polymorphic functions. The proposed compilation approach is possible in Poly C because of its close correspondence with Kernighan and Ritchie C and its robust type system. The type inference system of Poly C can be used to determine which functions are polymorphic. Poly C's type system assures type correctness of a Poly C program. Intermediate code generation can be based on type information determined during the type checking process.

The specific assumptions necessary for the approach proposed in this paper are:

- the compiler can determine if a function is monomorphic or polymorphic,
- the compiler can ascertain information required by each polymorphic function (array size, element size, pointer arithmetic to apply, etc.) and is able to supply the information.

The details of the approach are described in the next section. Both the concepts and benefits of the approach are presented. In Section 3, examples are used to demonstrate the technique.

2. Description of Approach

There are four basic kinds of functions in an imperative language such as Poly C:

- monomorphic applicative -- a monomorphic function executed for its value,
- monomorphic imperative -- a monomorphic function executed for its effect and perhaps a value,
- polymorphic applicative -- a polymorphic function executed for its value,
- polymorphic imperative -- a polymorphic function executed for its effect and perhaps a value.

The first two kinds of functions when written in Poly C can be translated directly into C code. Polymorphic functions must be handled differently to allow the same code to be

executed for all data types and produce correct results. This paper concentrates on intermediate code generation for the two kinds of polymorphic functions. To obtain executable code, each of the two kinds of polymorphic functions must be handled differently during compilation.

To correctly generate intermediate code from a Poly C program, the compiler must first determine the kind of function being translated. This is possible with information produced by Poly C's type inference system. Specifically,

- If the return type of the function is quantified (non-specialized) then the function is polymorphic.
- If the return type of the function is specialized then the function is monomorphic.
- If the function has no return type then it is executed for its effect.

A standard convention is always used to pass data to and from a polymorphic function. We use a calling convention employing a uniform representation. The types of the called function's actual parameters are ignored. The same number of bytes are allocated on the function's activation stack for each parameter regardless of its type.[1] In the context of this paper, uniform representation refers to the notion that parameters occupy a uniform size on a polymorphic function's stack.

For example consider the polymorphic identity function, called **id**, written in Poly C and the associated intermediate code present below in Figure 1.

Poly C Program	Intermediate Code

letvar one :=
  let **id** = λx.x in //polymorphic function id
in **id**(1)
	long **poly_id**(long x) {return x;} // function id
	
	int main()
	{
	  int one = **poly_id**(1);
	  return 0;
	}

Figure 1. Polymorphic Identity Function.

[1] The calling convention of typical C compilers passes arguments to functions on the stack and passes a return value from the function in a specific register(s).

A default size of four bytes (type *long*) is used in the intermediate code for id's argument and return-value. The actual default size chosen for the implementation depends of the target architecture. In this paper, type *long* is used in the intermediate code for all examples of polymorphic functions. After compilation, the C program executes correctly for all data types currently supported in Poly C. Test results of the associated intermediate code for the Poly C program of Figure 1 are given in Appendix A.

As shown above, when a polymorphic function is executed for its value, an implementation using a calling convention employing default size for formal parameters on the activation stack is sufficient to produce correct results. However if a polymorphic function is executed for its effect, a different approach and additional information is required. The intermediate code generated performs a byte-by-byte manipulation of the effected store. To do this correctly the size (number of bytes) of actual parameters must be passed to the function. In this situation the compiler must be able to determine the size of the data type being passed to the function. The assignment function presented below in Figure 2 is an example of byte-by-byte copy.

Poly C Program	Intermediate Code

let assign = λ l_side, r_side.(*l_side := r_side)
in ...
	void **poly_assign**(char *l_side, char *r_side, int size)
	{
	  int i = 0;
	
	  for(; i < size ; i++)
	  *(l_side + i) = *(r_side + i);
	}

Figure 2. A Polymorphic Function Executed for its Effect.

The intermediate code uses a type *char* pointer to facilitate byte-wise manipulation. The **for loop** copies each byte of actual parameter *r_side* to the appropriate byte of actual parameter *l_side*. The loop exit condition is based on the size of data type being manipulated. Test results of the associated intermediate code for the Poly C program of Figure 2 are given in Appendix B.

Another situation that occurs with polymorphic functions is that a specialized function may need to be passed to a polymorphic function. As in the case of a polymorphic sort function, the comparison function specific to the data types being sorted, must be supplied.

As currently described, only three data types are of concern for polymorphic function calls in Poly C: integers, pointers, and arrays. However array names function as constant pointers as in C, so really only two data types must be considered. In this paper various data types are used in the examples. The types are chosen to provide a range of store size and interpretation to sufficiently demonstrate the proposed implementation approach.

The primary advantage of this approach is that only polymorphic functions are affected by the proposed implementation. The impact on speed and storage of a Poly C program is minimal. Most of a Poly C's data type representation is unrestricted allowing storage optimization for all data values except parameters and return values of polymorphic functions. The only significant slow down potentially occurs when byte-wise manipulation is used in polymorphic functions executed for effect.

3. Demonstration of Approach

This section demonstrates the viability of implementing Poly C using the ideas presented above (default size occupied on the stack and byte-wise copy). Various conditions related to polymorphic functions that may occur in a Poly C program are explored. These conditions involve parameter passing combinations, functions executed for their value and functions executed for their effect, and continuation combinations[2].

[2] Chained function calls such as f1(f2(f3(x))).

In [MDCB91] four combinations of parameter passing encountered in polymorphism are discussed.

1. A concrete (specialized) actual parameter passed to a concrete formal parameter.
2. A concrete actual parameter passed to a quantified (non-specialized) formal parameter.
3. A quantified actual parameter passed to a concrete formal parameter.
4. A quantified actual parameter passed to a quantified formal parameter.

For the approach presented in this paper, these cases are not of primary interest since in Poly C all actual parameters are concrete[3].

A parameter passing combination that is of interest is the situation where a function is passed as a parameter to a polymorphic function. In the example given below in Figure 3 a polymorphic function called **apply** is passed two parameters, a data value and a function. The actual parameter for the function is passed via a pointer. Test results of the associated intermediate code for the Poly C program of Figure 3 are given in Appendix C.

Poly C Program	Intermediate Code

letvar succ_of_x =
  let apply = λx.λf.fx in
  let int_succ = λa.a + 1
in apply(2, int_succ)
	long **poly_apply**(long x, void (*fun)()) { return fun(x);}
	int **int_succ**(int a) {return a + 1;}
	int main()
	{
	  int x = 2;
	  int succ_of_x;
	  succ_of_x = **poly_apply**(x, **int_succ**);
	
	  return 0;
	}

Figure 3. A Function Passed as a Parameter to a Polymorphic Function.

The examples given in Figure 1 and Figure 3 involved functions executed for their value. When a function is executed for its effect a byte-wise copy is used to change the

[3] A polymorphic function passed as an actual parameter is concrete and has type *pointer*.

content of the appropriate memory locations. In this situation, the size of the data value being manipulated must be passed to the polymorphic function. This was demonstrated in the assignment example in the previous section (Figure 2).

The continuation of function calls (chaining of functions) can occur in four combinations.

1. A monomorphic function calls a monomorphic function.
2. A monomorphic function calls a polymorphic function.
3. A polymorphic function calls a monomorphic function.
4. A polymorphic function calls a polymorphic function.

The first calling combination does not involve any polymorphic functions and is not demonstrated. Also combination number three, a polymorphic function (**apply**) calls a monomorphic function (**int_succ**) was demonstrated above in Figure 3. Examples for each of the other two combinations are given below in Figure 4 and Figure 5. The Poly C program in Figure 5 is from [SmVo95]. Test results of the associated intermediate code for the Poly C programs of Figure 4 and Figure 5 are given in Appendix D and Appendix E respectively.

Poly C Program	Intermediate Code

letvar a = 1234 in
letvar copy_of_a = 0 in
let id = λx.x in
let int_id = λx.id(x) in
copy_of_a := int_id(a)
	// polymorphic id function
	long **poly_id**(long x) {return x;}
	
	// integer id function calls **poly_id**
	int **int_id**(int x) {return **poly_id**(x);}
	
	int main()
	{
	  int a = 1234;
	  int copy_of_a = 0;
	
	  copy_of_a = **int_id**(a);
	  return 0;
	}

Figure 4. A Monomorphic Function Calls a Polymorphic Function.

Poly C Program	Intermediate Code

let swap = λx,y.letvar t := *x;
in *x := *y; *y := t
in
let reverse = λa, n.letvar i := 0 in
while i < n - 1 -i do
swap(a + i, a + n - 1 - i);
i := i + 1
in ...
	// function to swap two elements
	void **poly_swap**(char* x, char* y, int size)
	{
	char temp;
	int i;
	for(i = 0; i < size; i++) {
	temp = *(x + i);
	*(x + i) = *(y + i);
	*(y + i) = temp;
	}
	}
	
	//function to reverse the elements of an array
	void **poly_rev**(char* x, int size, int n)
	{
	int i = 0;
	
	while(i < n - 1 - i) {
	poly_swap(x + (size * i), x + (size * (n - 1 - i)),
	size);
	i++;
	}
	}

Figure 5. A Polymorphic Function Calls a Polymorphic Function.

4. Conclusions

This paper demonstrated that intermediate C code can be used to efficiently implement polymorphism in Poly C.

There is still much to be done related to the approach presented in this paper.

- A translation scheme must be developed.
- Comparing the efficiency of this approach with other approaches.
- The method will need to be extended to more complicated data types as they are include in Poly C.

Although a translation scheme was not provided, the examples presented in Section 3 can serve as a guide for the development of formal methods for generating intermediate code. Various implementation methods for Poly C ([Bon95], the approach

presented in this paper, etc.) should be benchmarked for speed and space efficiency and compared with each other. From the perspective of intermediate code generation, extending Poly C to include more complicated data types does not appear to be an issue. The results of the examples presented in the appendices demonstrate how pointers can be used to handle other data types such as floats and structures.

One issue is the warnings generated by the compiler when the intermediate code is compiled. The type checking done by the C compiler when generating the executable program gives type warnings that can be ignored for the most part. Translating directly to assembly code would alleviate the problem.

APPENDIX A

Polymorphic Identity Function

Executable Code

```
1   #include <stdio.h>
2
3   long poly_id(long x) {return x;}
4
5   int main()
6   {
7     int a = 1234;
8     long b = 7777777;
9     int* ptr_a = &a;
10    int* c;
11
12    printf("\n *** testing poly_id ******\n\n");
13    printf(" %d", poly_id(a)); printf("\n");
14    printf(" %ld", poly_id(b)); printf("\n");
15    c = poly_id(ptr_a);
16    printf(" %d", *c );   printf("\n");
17
18    return 0;
19  }
```

Test Results

```
*** testing poly_id ******

1234
7777777
1234
```

Warning

```
line number: (15) : warning: 'argument' : different levels of indirection
line number: (15) : warning: 'poly_id' : different types for formal and actual parameter 1
line number: (15) : warning: '=' : different levels of indirection
line number: (15) : warning: conversion of near pointer to long integer
```

APPENDIX B

Polymorphic Assignment Function

Executable Code

```
1   // assignment function for Poly C
2
3   #include <stdio.h>
4
5   //test data
6   int x = 3;
7   int y = 44;
8
9   int x1 = 1;
10  int x77 = 77;
11
12  int *ptr_x1 = &x1;
13  int *ptr_x77 = &x77;
14  int m; // used for displaying contents of pointer
15
16  int arr_1[3] = {11, 12, 13};
17  int arr_2[3] = {21, 22, 23};
18
19  // polymorphic assignment function
20  // copies bytes from first parameter (l_side) to
21  // second parameter (r_side)
22  void poly_assign(char *l_side, char *r_side, int size)
23  {
24    int i = 0;
25
26    printf("\n poly_assign called.\n");
27    printf("   Size of l_side is ");  printf("%d", size); printf("\n");
28
29    for(; i < size ; i++)
30      *(l_side + i) = *(r_side + i);
31  }
32
33  // displays the elements of an array
34  void display_int_arr(int *p, const int num_of_things);
35
36  int main()
37  {
38    // testing with an integer
39    printf("\n ***** Testing assignment with an integer *****\n");
40    printf("\n The value of x is "); printf("%d ", x);
41    printf(" The value of y is ");  printf("%d ", y); printf("\n");
42
43    poly_assign(&x, &y, sizeof(x));
```

```
44
45      printf("\n The value of x is "); printf("%d ", x);
46      printf(" The value of y is ");  printf("%d ", y); printf("\n");
47
48      // testing with a pointer
49      printf("\n ***** Testing assignment with a pointer *****\n");
50      printf("\n The value of x1 is "); printf("%d ", x1);
51      printf("\n The value of x77 is "); printf("%d ", x77);
52      printf("\n");
53      printf("\n The value of ptr_x1 is "); printf("%d ", ptr_x1);
54
55      m = *ptr_x1;
56      printf("\n The value pointed to by prt_x1 is "); printf(" %d", m);
57      printf("\n The value of ptr_x77 is "); printf("%d ", ptr_x77);
58      printf("\n");
59
60      poly_assign(&ptr_x1, &ptr_x77, sizeof(ptr_x1));
61
62      printf("\n The value of ptr_x1 is "); printf("%d ", ptr_x1);
63
64      m = *ptr_x1;
65      printf("\n The value pointed to by prt_x1 is "); printf(" %d", m);
66      printf("\n The value of ptr_x77 is "); printf("%d ", ptr_x77);
67      printf("\n");
68      printf("\n The value of x1 is "); printf("%d ", x1);
69      printf("\n The value of x77 is "); printf("%d ", x77); printf("\n");
70
71      // testing with an array
72      printf("\n ***** Testing assignment with an array *****\n");
73      printf("\n The elements in array arr_1 are ");
74      display_int_arr(arr_1, 3);
75      printf("\n The elements in array arr_2 are ");
76      display_int_arr(arr_2, 3); printf("\n");
77
78      poly_assign(&arr_1, &arr_2, sizeof(arr_1));
79
80      printf("\n The elements in array arr_1 are ");
81      display_int_arr(arr_1, 3);
82      printf("\n The elements in array arr_2 are ");
83      display_int_arr(arr_2, 3); printf("\n");
84
85  return 0;
86  }
87
88  void display_int_arr(int *p, const int num_of_things)
89  {
90    int i = 0;
91    for(; i < num_of_things; i++)
92      printf("%d ", *(p + i));
93  }
```

Test Results

```
***** Testing assignment with an integer *****
The value of x is 3  The value of y is 44

poly_assign called.
  Size of l_side is 2

The value of x is 44  The value of y is 44

***** Testing assignment with a pointer *****
The value of x1 is 1
The value of x77 is 77

The value of ptr_x1 is 20
The value pointed to by prt_x1 is  1
The value of ptr_x77 is 22

poly_assign called.
  Size of l_side is 2

The value of ptr_x1 is 22
The value pointed to by prt_x1 is  77
The value of ptr_x77 is 22

The value of x1 is 1
The value of x77 is 77

***** Testing assignment with an array *****
The elements in array arr_1 are 11 12 13
The elements in array arr_2 are 21 22 23

poly_assign called.
  Size of l_side is 6

The elements in array arr_1 are 21 22 23
The elements in array arr_2 are 21 22 23
```

Warnings

```
line number: (43) : warning: 'argument' : indirection to different types
line number: (43) : warning: 'argument' : indirection to different types
line number: (60) : warning: 'argument' : different levels of indirection
line number: (60) : warning: 'poly_assign' : different types for formal and actual parameter 1
line number: (60) : warning: 'argument' : different levels of indirection
line number: (60) : warning: 'poly_assign' : different types for formal and actual parameter 2
line number: (78) : warning: 'argument' : different levels of indirection
line number: (78) : warning: 'poly_assign' : different types for formal and actual parameter 1
line number: (78) : warning: 'argument' : different levels of indirection
line number: (78) : warning: 'poly_assign' : different types for formal and actual parameter 2
```

APPENDIX C

Monomorphic Function Passed as a Parameter to a Polymorphic Function

Executable Code

```
1   // Test of polymorphic apply function that calls
2   // a monomorphic function.
3
4   #include <stdio.h>
5
6   float greg;  // used for returning an address from function call
7   float* p_greg = &greg;
8
9
10  // polymorphic apply function
11  long poly_apply(long x, void (*fun)()) {fun(x);}
12
13  // integer successor function
14  int int_succ(int a) {return a + 1;}
15
16  // long integer successor function
17  long long_succ(long a) { return a + 1;}
18
19  // float function
20  float* float_succ(float* a)  // pointers used for float
21  {
22    greg = *a + 0.1;
23    return p_greg;
24  }
25
26
27  int main()
28  {
29    // test data
30    int x = 2;
31    long L = 345678;
32    float f = 1.23F;
33    float f_res = 4.4F;
34    float* f_ptr = &f_res;
35
36    // a call to an integer successor function
37    printf("*****  Testing call to a int_succ *****");printf("\n");
38    printf("\n the value of x is ");
39    printf("%d", x); printf("\n");
```

```
40      printf(" the value of poly_apply(x, int_succ) is ");
41      printf("%d", poly_apply(x, int_succ)); printf("\n\n");
42
43      // a call to a long integer successor function
44      printf("*****  Testing call to a long_succ *****");printf("\n");
45      printf("\n the value of L is ");
46      printf("%ld", L); printf("\n");
47      printf(" the value of poly_apply(L, long_succ) is ");
48      printf("%ld", poly_apply(L, long_succ)); printf("\n\n");
49
50      // a call to a float successor function
51      printf("*****  Testing call to a float_succ *****");printf("\n");
52      printf("\n the value of f is ");
53      printf("%f", f); printf("\n");
54      printf(" the value of poly_apply(f, float_succ)"
55       "\n [incremented by 0.1] is ");
56      f_ptr = poly_apply(&f, float_succ);
57      f_res = *f_ptr;
58      printf("%f", f_res); printf("\n\n");
59
60      return 0;
61  }
```

Test Results

```
*****  Testing call to a int_succ *****
 the value of x is 2
 the value of poly_apply(x, int_succ) is 3

*****  Testing call to a long_succ *****
 the value of L is 345678
 the value of poly_apply(L, long_succ) is 345679

*****  Testing call to a float_succ *****
 the value of f is 1.230000
 the value of poly_apply(f, float_succ)
 [incremented by 0.1] is 1.330000
```

Warnings

```
line number: (11) : warning: 'poly_apply' : no return value
line number: (22) : warning: conversion between different floating-point types
line number: (56) : warning: 'argument' : different levels of indirection
line number: (56) : warning: 'poly_apply' : different types for formal and actual parameter 1
line number: (56) : warning: 'argument' : different levels of indirection
line number: (56) : warning: 'poly_apply' : different types for formal and actual parameter 2
line number: (56) : warning: '=' : different levels of indirection
line number: (56) : warning: conversion of near pointer to long integer
```

APPENDIX D

A Monomorphic Function Calls a Polymorphic Function

Executable Code

```
1   // monomorphic id functions call polymorphic id function
2
3   #include <stdio.h>
4
5   /*
6   //poly id using assmbly code
7   long asm_id(long x)
8   {
9     __asm mov ax, [bp+6]
10    __asm mov dx, [bp+8]
11  }
12  */
13
14
15  // poly id using C code
16  long poly_id(long x) {return x;}
17
18  // monomorphic functions
19  int int_id(int x)
20  {
21    return poly_id(x);  // poly_id called
22  }
23
24  long long_id(long x)
25  {
26    return poly_id(x);  // poly_id called
27  }
28
29  int* ptr_id(int* x)
30  {
31    return poly_id(x);  // poly_id called
32  }
33
34
35  int main()
36  {
37    int a = 1234;
38    long b = 7777777;
39    int* ptr_a = &a;
40    int* c;
41
```

```
42      printf("\n *** testing id ******\n\n");
43      printf(" %d", int_id(a)); printf("\n");
44      printf(" %ld", long_id(b)); printf("\n");
45      c = ptr_id(ptr_a);
46      printf(" %d", *c );   printf("\n");
47
48      return 0;
49
50  }
```

Test Results

```
*** testing id ******

1234
7777777
1234
```

Warnings

```
line number: (21): conversion between different integral types
line number: (31): 'argument' : different levels of indirection
line number: (31): 'poly_id' : different types for formal and actual parameter 1
line number: (31): 'return' : different levels of indirection
line number: (31): conversion of near pointer to long integer
```

APPENDIX E

A Polymorphic Function calls a Polymorphic Function

Executable Code

```
1   // polymorphic reverse function.
2
3   #include <stdio.h>
4
5   // polymorphic function to swap two elements
6   void poly_swap(char* x, char* y, int size)
7   {
8     char temp;
9     int i;
10    for(i = 0; i < size; i++) {
11      temp = *(x+i);
12      *(x+i) = *(y+i);
13      *(y+i) = temp;
14    }
15  }
16
17  // function that reverses the elements of an array
18  // element size required for indexing the array
19  void poly_rev(char* x, int size, int n)
20  {
21    int i = 0;
22
23    while(i < n-1-i) {
24      poly_swap(x+(size*i), x+(size*(n-1-i)), size);
25      i++;
26    }
27  }
28
29
30  int main()
31  {
32    int int_arr[4] = {1, 2, 3, 4};
33    long long_arr[4] = {44444, 55555, 66666,77777};
34    float float_arr[4] = {1.1F, 2.2F, 3.3F, 4.4F};
35    int a = 10;
36    int b = 11;
37    int c = 12;
38    int d = 13;
39    int* ptr_arr[4] = {&a, &b, &c, &d};
40    int i;
41
```

```
42      printf("\n *** testing poly_reverse ******\n\n");
43
44      // int_arr
45      printf(" int_arr prior to reversal  ");
46      for(i = 0; i < 4; i++)
47        printf("%d ", int_arr[i]);
48      poly_rev(int_arr, sizeof(*int_arr), 4);
49      printf("\n int_arr after reversal  ");
50      for(i = 0; i < 4; i++)
51        printf("%d ", int_arr[i]);
52
53      //long_arr
54      printf("\n\n long_arr prior to reversal ");
55      for(i = 0; i < 4; i++)
56        printf("%ld ", long_arr[i]);
57      printf("\n");
58      poly_rev(long_arr, sizeof(*long_arr), 4);
59      printf(" long_arr after reversal ");
60      for(i = 0; i < 4; i++)
61        printf("%ld ", long_arr[i]);
62
63    //float_arr
64      printf("\n\n float_arr prior to reversal ");
65      for(i = 0; i < 4; i++)
66        printf("%f ", float_arr[i]);
67      printf("\n");
68      poly_rev(float_arr, sizeof(*float_arr), 4);
69      printf(" float_arr after reversal ");
70      for(i = 0; i < 4; i++)
71        printf("%f ", float_arr[i]);
72      printf("\n");
73
74      //pointer_arr
75      printf("\n\n ptr_arr prior to reversal ");
76      for(i = 0; i < 4; i++)
77        printf("%d ", *ptr_arr[i]);
78      printf("\n");
79      poly_rev(ptr_arr, sizeof(*ptr_arr), 4);
80      printf(" ptr_arr after reversal ");
81      for(i = 0; i < 4; i++)
82        printf("%d ", *ptr_arr[i]);
83      printf("\n"); printf("\n");
84
85      return 0;
86
87    }
```

Test Results

```
*** testing poly_reverse ******

int_arr prior to reversal  1 2 3 4
int_arr after reversal  4 3 2 1

long_arr prior to reversal 44444 55555 66666 77777
long_arr after reversal 77777 66666 55555 44444

float_arr prior to reversal 1.100000 2.200000 3.300000 4.400000
float_arr after reversal 4.400000 3.300000 2.200000 1.100000


ptr_arr prior to reversal 10 11 12 13
ptr_arr after reversal 13 12 11 10
```

Warnings

```
line number: (48): 'argument' : indirection to different types
line number: (58): indirection to different types
line number: (68): indirection to different types
line number: (79): different levels of indirection
line number: (79): different types for formal and actual parameter 1
```

REFERENCES

[Bon95] Bonem, P., *Towards an Implementation of Polymorphic C*, Master's Thesis, Naval Postgraduate School, Monterey California, September 1995.

[Ler92] Leroy, S., "Unboxed Obejects and Polymorphic Typing", *Proc. 19th ACM Symposiun on Principles of Programming Languages*, January 1992.

[MDCB91] Morrison, R., Dearle, A., Connor, C., and Brown, A., "An Ad Hoc Approach to the Implementation of Polymorphism", *ACM transactions on Programming Languages and Systems*, Vol. 13, No. 3, July 1991.

[ShAp95] Shao, Z, and Appel, A., "A Typed-Based Compiler for Standard ML", *Proc. 1995 Conf. on Programming Language Design and Implementation*, June 1995.

[SmVo95] Smith, G., and Volpano, D., "An ML-style Polymorphic Type System for C", submitted for publication, 1995.

[Thi95] Thiemann, P., "Unboxed Values and Polymorphic Typing Revisited", *Conference Record of ACM FPCA '95 Conference on Functional Programming Languages and Computer Architecture*, June 1995.

DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
Library, Code 52 Naval Postgraduate School Monterey, CA 93943	2 copies
Center for Naval Analyses 4401 Ford Avenue Alexandria, VA 22302-0268	2 copies
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1 copy
Dr. D. Volpano, Code CS/Vo Computer Science Department Naval Postgraduate School Monterey, CA 93943	5 copies
Dr. G. Smith School of Computer Science Florida International University University Park Miami, FL 33199	1 copy
CDR Carl M. Pederson Jr., USN PSC 78 BOX 346 APO AP 96326-0346	1 copy